POOL-ORA approach to offline calibration/alignment (Draft)

May 30, 2005

1 Introduction

The tabular representation of data in a relational database is fundamentally different from the networks of objects used in object-oriented application. This difference is the so-called object/relational paradigm mismatch[3]. Object/Relational Mapping(ORM) is the automated persistency of objects in a domain application to the tables in a relational database. In essence, ORM works by transforming data from one presentation to another.

POOL[1] is the LHC object persistency framework in C++. There are two implementations of the persistency layer: ROOT/IO and RDBMS. In the RDBMS implementation, Object Relational Access(ORA) is the middleware in the persistency layer that manages the ORM. POOL-ORA implements ORM rules to bridge the object and the relational world. POOL uses XML as the format for describing the mapping metadata. If no user-defined XML mapping metadata is present, the default mapping rules will take effect.

POOL-ORA is a non-intrusive object persistency solution: users aren't required to follow specific rules when designing persistent classes thus it integrates smoothly with most new and existing applications and don't require disruptive changes in those application; users aren't required to design the database schema either since it is automatically generated by the ORM. POOL uses SEAL reflection[2] for interaction between persistency layer and transient application at run time. Reflection is the ability of the object to see the layout of itself at run time.

The following sections explain the POOL-ORA approach for CMS offline calibration and alignment data in a top-down fashion: from the object model to the database.

2 software process and maintenance

At build time, dictionary files are generated from header files of the desired objects, just as in POOL ROOT/IO object persistency which is used for the CMS event data.

At run time, the user asks for these objects and POOL delivers them. The application uses ORA through the generic POOL storage service interface specifying the storage type as "POOL_RDBMS". The users are not aware of the ORA machinery. The dictionary library is loaded for reflection, just as in POOL ROOT/IO.

The interface and the machinery for the conditions data which uses RDBMS, and the event data which uses POOL ROOT/IO is the same. There is no maintenance overhead of the conditions data with respect to the event data at the software layer.

3 Transaction model and caching

3.1 Transaction and first level caching

POOL has a well-defined transaction model. It distinguishes application and database level transactions. There is a first level in-memory cache attached to the application level transaction. The transactional-scope object identifier is the pool::Ref. The first-level cache is mandatory and can't be turned off. Two lookups using the same object identifier in the same session from two concurrently running database transactions result in the same object instance in the object cache.

3.2 Possibility and benefits of the secondary caching

The second-level cache sits between the domain application and the database. It makes data retrieved from the database in one application visible to another application. Since it caches the data across the application boundary,

it is also known as the process-scoped or cluster-scoped cache. In contrast to the first-level cache, the second-level cache stores database states, i.e. byte-stream, instead of navigable persistent instances. The object identifier in the second-level cache is different from that in the first-level cache because it should have a lifespan longer than a process and can be replicated in different machines in a cluster. The so-called *reference data*, which is rarely updated and referenced by many instances of other classes,is an excellent candidate for second-level caching and any application that uses *reference data* heavily will benefit greatly if that data is cached.

Though currently not providing a second-level cache, POOL implements long-lived object identifiers, pool::Token, which makes second-level process or cluster scope caching possible.

4 Offline calibration and alignment data model

For CMS offline calibration and alignment software, the proposed objects are shown in the Appendix. These objects are seen by calibration and alignment applications and required to be persistable, in other words, can be stored as tables in the database.

The interval of validity (IOV) and the IOV metadata objects, which are responsible for managing and ensuring the data validity and conditions, are part of the data model. These objects are hidden from the physics applications but are nevertheless required to be persistable.

In the POOL-ORA approach, the transient object model in the application drives the database data model. The designers of the object model don't need to know the tabular representation of the data in the database. The database schema is automatically created from the definitions of the persistable objects. How POOL-ORA achieve this is described in the following section.

5 ORM and offline database schema

5.1 ORM rules

The database data schema is defined by the application object model. The database schema is automatically generated by object relational mapping in

POOL following the ORM rules as the following:

- basic class mapping and object identifier

  The rule for mapping basic classes, which contain only primitive data member and no referencing and inheritance relationships with other classes, is a simple "one table for every class" for POOL-ORA. The object id can be "native" which means that it is generated by POOL internally or it can be mapped to a primary key column of an existing table. The second case is useful in handling legacy relational data.

- embedded classes
  A class embedded in another class has no database identity. The persistent state of the embedded class is a subset of the table row of the owning class.

- class inheritance mapping
  There are three different approaches to representing an inheritance hierarchy:

  1. One table per concrete class
     Discard polymorphism and inheritance, each concrete class is mapped to a table.
  2. One table per class hierarchy
     Each class hierarchy is mapped to a table using a type discriminator column to hold type information. The relational model is denormalized.
  3. One table per subclass
     The inheritance relationships are represented as relational foreign key associations. Every subclass that declares persistent properties has its own table. Unlike the one table per concrete class strategy, the subclass table contains columns only for each non-inherited property along with a primary key that is also a foreign key of the superclass table.

  The default rule is "one table per concrete class".

- association mapping
  The entity association mapping is unidirectional – the behavior of a

non-persistent instance is the same as the behavior of a persistent instance.

5.2 Schema evolution and ORM versions

In the ORA approach, object data are stored/retrieved following the SEAL dictionary information and then finding the corresponding entries in the OR mapping files. The ORM is versioned and the version is specified in the mapping file. The ORM version is stored together with the object table. Many schema evolution cases can be treated transparently through this mechanism.

Changing in the transient shape of the object is defined by the object header file through the dictionary which has no effect on the database schema. Changing in the mapping rules, thus the database schema, is traced and controlled by the XML mapping file and the versioning of the mapping. Since the mapping is stored together with the objects one can switch between underlying schemas for the same objects transparently.

5.3 Number of tables for CMS calibration and alignment data model

The number of tables need to be managed depends on the object data model and the mapping rules applied to the model, in most cases is "one table per class", plus several tables of overhead for object management.

Take the calibration and alignment data model for example. There are mainly three persistable classes in the alignment task: Alignments, Geom2index and IOV. These classes contain non-trivial data structures such as maps, maps of int to vector of strings etc. By applying the default mapping rules, the alignment data model results in 7 tables plus 8 constant overhead of POOL container and mapping management tables. Considering the calibration objects, as shown in the Appendix, has a similar structure and share the IOV object with the alignment, one adds 5 tables for the calibration object which would make 20 tables sufficient for rudimental offline calibration and alignment tasks.

In a more elegant model, one might want to add metadata, such as tag, version, etc, to manage the IOV objects which would add 2 or 3 tables more for this purpose. However, it is possible that this kind of metadata infor-

mation will be managed by the metadata system rather than the calibration and alignment system.

By choosing non-default mapping rules, one might get even less number of tables.

5.4 Development processes and ORM toolsets

POOL-ORA applications can be run with default ORM rules without relying on user-customized mapping rules. However, for complex object model and sychronisation with legacy pure relational data, external mapping tools are usually required by the user. POOL-ORA comes bundled with a set of tools.

In the "Top down" development scenario where one starts with existing transient data model and complete freedom with respect to the database schema, tools are provided to create a mapping document manually or automatically from domain application source, then use another tool to generate the database schema.

In the other direction, tools for generating mapping documents from a POOL-ORA database schema are available. While tools for generating mapping documents from a legacy database schema are under development.

5.5 Online and Offline database data transfer

POOL-ORA based tools will be needed to extract and transfer the data from the online to the offline database. Since the OR mapping in POOL is driven by XML files, the same XML files will drive the online to offline transfer.

It is forseen that the offline applications will write directly to the offline databases although this data will not be forwarded to the online database.

6 Performance related aspects

POOL comes with an in-memory object cache which has a lifespan of an application session which means if one object is requested more than once in the same session, only the first request triggers a database transaction.

In the case of different processes asking for the same objects concurrently to the same database server, one would benefit in read-only operations if there were a secondary cache between the application and the database server.

From the database schema standpoint, one can tune the schema by choosing different mapping rules. From the server side, indexes can be added to the schema once the access pattern of an object or an object graph is well-understood and the bottom-neck of the database access is clearly identified.

From the database connectivity layer RAL, both CMS and ATLAS has been making studies of POOL-RAL performance overhead over the raw database connectivity, so far these studies haven't reported worrying performance problem caused by the RAL layer neither in writing nor in reading.

7 Database deployment and maintenance

7.1 Database backends and deployment

In the overall POOL architecture, POOL-ORA is a layer on top of the POOL-RAL (Relational Abstract Layer) which is a SQL-free abstract layer to hide different RDBMS technologies from the user. Currently, POOL-RAL supports both native OCI binding to the Oracle database, native binding to sqlite database, as well as the generic ODBC connectivity to all the RDBMS backends(including ORACLE) which can be connected through ODBC drivers. The ORM mapping files do not expose any SQL syntax, thus are technology neutral. Therefore, POOL-ORA is a RDBMS technology neutral persistency solution.

There is consensus that ORACLE database requires much steeper learning curve than other RDBMS for maintenance and management. Since POOL-ORA is technology neutral, this flexibility allows the deployment of the offline conditions and alignment database in other technologies, such as mysql, sqlite,etc for smaller and/or local sites while the database backend differences are completely hidden from the application software. In the meantime, the data can be transfered between the different database backends using POOL-RAL.

7.2 DBA tasks

As described above, the POOL-ORA approach follows the "top-down" development process, i.e. object model driven. There is not only no need for explicit ORACLE schema design nor is there for other RDBMS. Data can be

transfered between databases transparently through POOL-RAL. Such a system requires much less database maintenance effort with respect to a system developed from "bottom-up", which is database schema driven, particularly if more than one RDBMS technology is desired.

Nevertheless, POOL-ORA uses databases, inevitably there are database maintenance and DBA issues. These tasks are decided by the database deployment model which therefore should be defined by the CMS computing model together with the LCG Distributed Database Deployment(3D) project[4]. The POOL-ORA approach itself requires almost no database design and maintenance tasks.

There are two distinct DBA roles as in any database project maintenance:

- Basic, mission-critical DBA role

  Each tier will have DBA experts working with the LCG 3D Project to determine what tables will be replicated and how this will be accomplished, for example using Oracle Streams for the major sites (Tier 0, Tier 1 and possibly some Tier 2).

  CERN IT database service group will provide fundamental DBA skills for setup, maintenance and backup of high-availability ORACLE servers for the CMS Tier 0. It is assumed that the same basic services will be provided at each large site with the explicit assumption that these sites' DBAs will work in a coordinated effort to ensure the integrity and availability of the conditions and calibration data.

- Developer DBA role

  This role provides DBA intelligence and rationale for design of the database by monitoring client and server performance of a database application and help the software developer to locate flaws in the design.

  Take POOL-ORA application for example, if one application is reported not to fulfill the performance requirement, the developer DBA should analyze the database access pattern, client-server traffic etc and come up with a recommendation, e.g. change mapping rules or switch on secondary caching for a particular object or a graph of objects.

Appendix

```
class  AlignTransform {
  public:
    typedef CLHEP::HepEulerAngles EulerAngles;
    typedef CLHEP::Hep3Vector ThreeVector;
    typedef CLHEP::HepRotation Rotation;
    typedef ThreeVector Translation;

    AlignTransform();
    AlignTransform( const Translation & itranslation,
  const EulerAngles & ieulerAngles);
    AlignTransform( const Translation & itranslation,
  const Rotation & irotation);
    const Translation & translation() const;
    const EulerAngles & eulerAngles() const;
    Rotation rotation() const;
  private:
    Translation m_translation;
    EulerAngles m_eulerAngles;
};

class Alignments {
  public:
    Alignments();
    virtual ~Alignments();
    std::vector<AlignTransform> m_align;
};

class Geom2Index {
  public:
    typedef int GeomId;
    typedef int Index;
    typedef std::map<GeomId,Index> IndexTable;
    typedef std::vector<GeomId> GeomIdTable;

    Geom2Index();
```

```
    virtual ~Geom2Index();

    IndexTable m_indexTable;
    GeomIdTable m_GeomIdTable;
};


class BaseCalib {
    public:
    virtual ~BaseCalib();
};

class Pedestals : public BaseCalib {
  public:
    struct Item {
      Item(){}
      float m_mean;
      float m_variance;
    };
    typedef int ElectronicsId;
    typedef std::vector<Item>::const_iterator ItemIterator;
    typedef std::pair<ItemIterator,ItemIterator> ItemRange;
    typedef std::map<ElectronicsId,std::vector<Item> > Map;
    typedef Map::const_iterator MapIterator;

    Pedestals();
    ItemRange get(ElectronicsId id) const;
    std::map<ElectronicsId,std::vector<Item> >  m_pedestals;
};

class Geom2Electronics : public BaseCalib {
  public:
    typedef int GeomId;
    typedef int ElectronicsId;
    typedef std::map<GeomId,ElectronicsId> Table;
    typedef Table::const_iterator TableIterator;
```

```
    Geom2Electronics();
    ElectronicsId operator[](GeomId id) const;
    const std::map<GeomId,ElectronicsId> & lookupTable() const;
    std::map<GeomId,ElectronicsId> m_lookupTable;
};
```

References

[1] **http://pool.cern.ch**

[2] **http://seal.cern.ch**

[3] Ambler, Scott, 2002. **"Mapping Objects to Relational Databases"**. AmbySoft Inc. white paper. www.ambysoft.com/mappingObjects.html.

[4] **http://lcg3d.cern.ch**